MINI CURSO DE PYTHON

FERNANDO FELTRIN

PYTHON NEXUS

MINI CURSO DE PYTHON

Todo o necessário para você criar seus primeiros programas em Python, em menos de 50 páginas de conteúdo!

FERNANDO FELTRIN

PYTHON NEXUS

AVISOS

Este é um material INTRODUTÓRIO sobre programação em Python, GRATUITO, sua revenda ou distribuição mediante qualquer tipo de pagamento é ilegal.

Este livro conta com mecanismo antipirataria Amazon Kindle Protect DRM. Cada cópia possui um identificador próprio rastreável, a distribuição ilegal deste conteúdo resultará nas medidas legais cabíveis.

É permitido o uso de trechos do conteúdo para uso como fonte desde que dados os devidos créditos ao autor.

PYTHON NEXUS

SOBRE O AUTOR

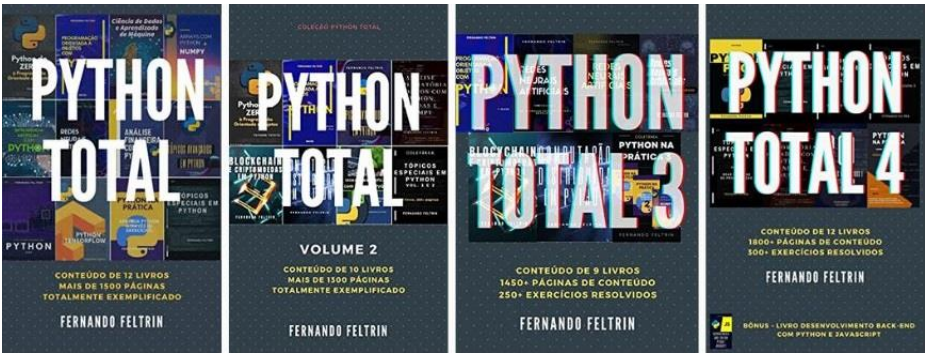
Fernando Feltrin é Engenheiro da Computação com especializações na área de ciência de dados e inteligência artificial, Professor licenciado para docência de nível técnico e superior, Autor de mais de 30 livros sobre programação de computadores e responsável pelo desenvolvimento e implementação de ferramentas voltadas a modelos de redes neurais artificiais aplicadas à radiologia médica (diagnóstico por imagem).

LIVROS

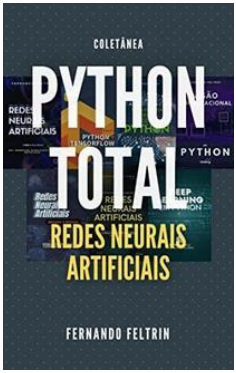
PYTHON NEXUS

Disponível em: [Amazon.com.br]

Coletâneas PYTHON TOTAL



PYTHON NEXUS



Disponível em: [Amazon.com.br]

PYTHON NEXUS

CURSO

[Curso Python do ZERO à Programação Orientada a Objetos]

Mais de 15 horas de videoaulas que lhe ensinarão programação em linguagem Python de forma simples, prática e objetiva.

ÍNDICE

INTRODUÇÃO

No mundo atual, graças a rápida evolução da tecnologia, a capacidade de se adaptar e aprender novas habilidades é um ingrediente essencial para quem busca se tornar um desenvolvedor. Neste contexto, a linguagem de programação Python emergiu como uma das ferramentas mais versáteis e poderosas tanto para iniciantes quanto para especialistas em programação.

Ao longo deste livro, você encontrará uma abordagem passo a passo para aprender os fundamentos da linguagem de programação Python, desde a instalação do ambiente de desenvolvimento até a construção de seus primeiros projetos.

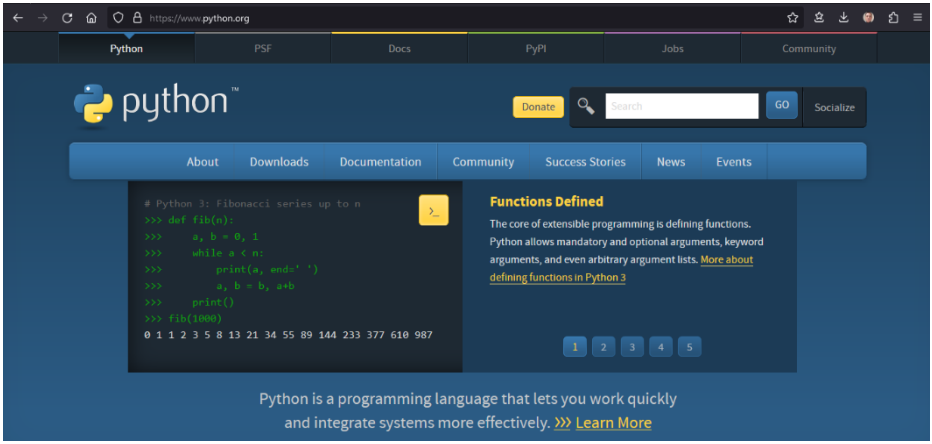
Para isso, o "Mini Curso de Python" tem como objetivo apresentar os conceitos fundamentais de maneira clara e simplificada, utilizando exemplos práticos que permitirão a você aplicar o que aprendeu imediatamente.

Então, prepare-se para embarcar nesta empolgante jornada e descobrir o potencial que a linguagem Python tem a oferecer. Seja você um estudante, um profissional ou simplesmente um entusiasta da tecnologia, este mini curso fornecerá toda a base necessária para que você dê seus primeiros passos desenvolvendo suas próprias aplicações.

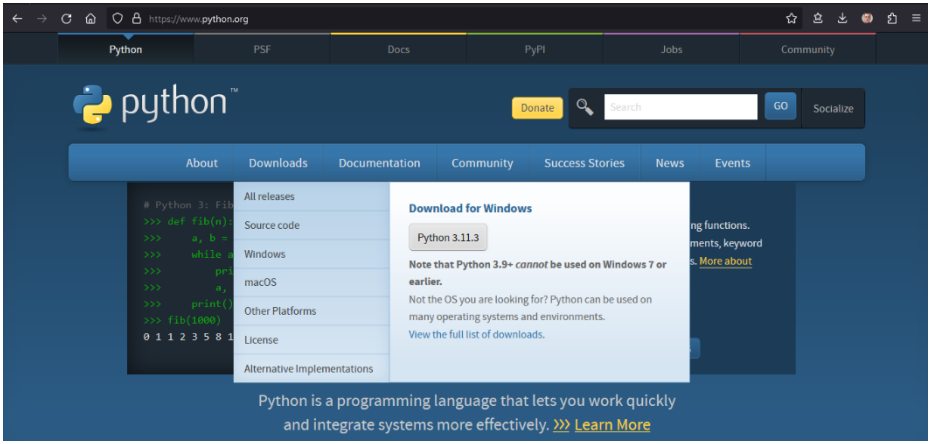
PREPARAÇÃO DO AMBIENTE

Como ponto de partida, se faz necessário entender que dependendo de seu sistema operacional (principalmente Windows), a linguagem Python pode não ser nativa, sendo necessário realizar sua instalação no núcleo do sistema, processo que é feito assim como para qualquer outro programa.

Para isso, vamos seguir alguns passos:



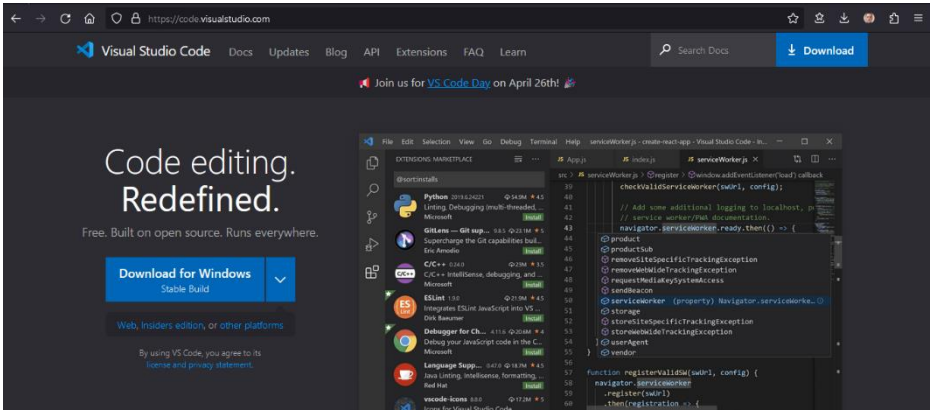
Primeiramente acessamos o site oficial da linguagem de programação Python python.org.



Em seguida, acessando a guia Downloads é aberto um menu flutuante com o link para a última versão estável da linguagem, neste caso, a versão 3.11.

Clicando no botão logo abaixo de Download for Windows, o download do instalador é inicializado. Caso queira instalar outra versão específica ou de outra arquitetura (32 bits por exemplo), basta acessar o link *View the full list of downloads.*

Após concluída a instalação da linguagem Python no sistema operacional, o próximo passo é instalar uma IDE. Basicamente, uma IDE é um editor de texto dotado de alguns recursos voltados ao desenvolvimento, uma ferramenta que nos permitirá não somente escrever nossos códigos mais testá-los em tempo real.



Existe uma vasta gama de IDEs disponíveis no mercado, cada uma com suas particularidades, vantagens e desvantagens. A nível de recomendação, sugiro o uso do Visual Studio Code, totalmente gratuito, disponível a partir do site visualstudio.com.

Em destaque na própria página inicial, temos o link para download da última versão estável deste ambiente de desenvolvimento integrado.

Uma vez que temos o núcleo da linguagem Python e uma IDE instalados em nosso sistema operacional, podemos dar início aos conceitos voltados à programação.

SINTAXE BÁSICA

Um dos primeiros pontos que temos de abordar quando o assunto é programação, independentemente da linguagem de programação a ser utilizada, é o que diz respeito à sua sintaxe.

Assim como quando nos alfabetizamos aprendemos os meios e métodos de escrita, desde letras, sílabas, palavras até conjunções complexas em frases para assim nos expressarmos de alguma forma, quando estamos aprendendo uma linguagem de programação o processo é praticamente o mesmo.

Porém, cabe salientar que, diferentemente de uma linguagem natural utilizada por humanos, onde temos capacidade de abstração para entender os conceitos de um texto, uma máquina apenas lê e executa instruções. Em função disso, devemos escrever exatamente o que a máquina espera, caso contrário a mesma não entenderá a instrução e não realizará nenhuma ação.

A chamada sintaxe de uma linguagem de programação é exatamente isto, a forma como escrever blocos de código (scripts de algoritmos) de modo que tais códigos sejam devidamente interpretados.

A linguagem Python é uma das linguagens de maior destaque por conta do que tange a sintaxe, pois possui uma sintaxe bastante simples e objetiva, utilizando muito de termos usuais em inglês para definir suas estruturas de código. Em

outras palavras, diferentemente de outras linguagens de programação, em Python com poucas linhas de código conseguimos criar aplicações inteiras.

Conforme veremos nos capítulos subsequentes, cada estrutura de código terá uma forma particular de escrita que utilizaremos sempre, independentemente do contexto.

Leitura Léxica

Outro ponto importante a entender neste momento é o conceito de leitura léxica, onde basicamente tratamos da forma como nossos códigos serão lidos e interpretados.

A linguagem Python é uma linguagem interpretada, e isto significa que, diferentemente de outras linguagens (compiladas) onde se escreve todo o código, depois se compila para finalmente poder de fato testar a aplicação, em Python temos um agente chamado interpretador, que consegue ler e executar instruções em tempo real.

Raciocine que por parte do interpretador, o mesmo possui como regra básica a leitura do código linha por linha, da esquerda para direita, fazendo as devidas relações entre as estruturas de código para que o programa realize suas funções. Isto implica que devemos seguir algumas regras de escrita para que nossos códigos de fato sejam funcionais, algo que veremos diretamente na prática nos próximos capítulos.

Indentação

Um último ponto a ser entendido antes mesmo de partirmos de fato para a prática é sobre a chamada “indentação”.

Basicamente, raciocine que complementar ao que foi dito anteriormente para a leitura léxica, a tabulação aplicada aos códigos escritos em Python também importa.

Em outras palavras, além da disposição sequencial em linhas para nossos códigos, os “parágrafos” aplicados neste contexto sinalizam ao interpretador a hierarquia das estruturas de código, haja visto que será perfeitamente normal encadear estruturas de código umas dentro das outras.

Todos estes conceitos logo farão sentido à medida que avançamos para os tópicos práticos deste livro.

VARIÁVEIS

Seguindo com nossa linha de raciocínio, um dos pontos chave a serem entendidos é o conceito de variáveis e sua utilização.

Para isso, vamos inicialmente fazer um simples exercício mental. Imagine em sua casa um móvel composto por algumas gavetas, o qual utilizaremos para guardar peças de roupas de modo organizado. Digamos que em uma destas gavetas temos apenas meias, toda vez que precisamos utilizar de um par de meias, abrimos tal gaveta e pegamos estes objetos, os usamos, lavamos e novamente os guardamos na mesma gaveta.

Uma variável é como uma gaveta de um móvel, onde estaremos guardando dados / valores / informações de modo que sempre que precisarmos fazer uso

destes dados, tal variável será acessada para que seu conteúdo seja lido e interpretado.

Dessa forma, enquanto nossas aplicações estiverem sendo executadas, dependendo de sua robustez, diversas variáveis estarão em uso, tendo seu conteúdo alocado e carregado em memória, para que possamos iterar sobre tais dados sempre que necessário.

Declarar uma variável em Python é um processo bastante simples, onde apenas devemos respeitar algumas simples regras de sintaxe para que tal estrutura funcione conforme o esperado. De acordo com a leitura léxica aplicada, uma variável deve obrigatoriamente possuir um nome, um operador de atribuição e um atributo (os próprios dados / valores associados a tal variável).

```
1   nome_da_variavel = 'dado / valor / atributo'
2
```

Partindo para a prática, eis que na imagem acima temos um exemplo básico de declaração de uma variável. Neste código, logo de início declaramos uma variável de nome nome_da_variavel que recebe como atributo um dado em formato textual.

Observe que por parte de sintaxe, a variável possui um nome próprio personalizado, seguido de um símbolo “ = “ e de um dado, neste caso, de formato textual graças a notação aplicada de um conjunto de caracteres entre “” aspas. Como veremos logo no capítulo seguinte, de acordo com esta forma de escrita os dados atribuídos a uma variável serão identificados em tipos / categorias de dados diferentes.

Ainda no que diz respeito à sintaxe, é importante salientar que existem certas regras específicas de como podemos nomear corretamente nossas variáveis, pois nesse contexto, existem certas restrições que devemos levar em conta.

```
nome_da_variavel = 'dado / valor / atributo'
nomedavariavel = 'dado / valor / atributo'
nomeDaVariavel = 'dado / valor / atributo'

```

No código acima, a maneira usual e correta de se declarar variáveis em Python. Note que em todas as variáveis as mesmas possuem um nome próprio, único, onde é permitido o uso de espaçadores (não espaços em branco) e números (desde que não estejam na posição inicial do nome da variável.

```
_variavel1 = 'Python'
variável = 2001
Nome = 'Fernando'
USUARIO = 'fernando2rad'

```

Neste outro exemplo, algumas formas de nomear variáveis que são permitidos mais não recomendados. Por exemplo, iniciar o nome de uma variável com o caractere " _ " é permitido, porém esta notação é uma notação avançada que configura se tal variável estará visível ou não para as funções do código.

No exemplo da segunda linha deste código temos uma variável nomeada onde em seu nome um dos caracteres possui um acento. Embora permitido, lembre-se que a linguagem natural americana não carrega acentos, por conta disto, pode ser que surjam certas incompatibilidades quando tal código for lido por alguma máquina configurada apenas para o idioma inglês.

Já no terceiro exemplo temos uma variável declarada de modo que seu nome inicia em letra maiúscula. Apesar de permitido, esta notação normalmente é reservada para nomear classes, no paradigma da programação orientada à objetos, algo que não é nosso foco neste momento.

Por fim, temos uma variável com nome composto inteiramente de caracteres maiúsculos, o que em outras linguagens de programação configura uma constante, algo que não existe em Python por conta da mesma ser uma linguagem dinamicamente tipada, tendo suas variáveis a propriedade de serem

modificadas sempre que necessário, diferentemente de constantes que como próprio nome sugere, são objetos imutáveis após declarados.

```
10_livros = 'Livro 1, livro 2, etc...'
2000 = '2000'
@variavel1 = 12
(variavel1) = 1987

```

Dando continuidade, no exemplo acima temos alguns exemplos de formas não permitidas, e quando falamos em algo não permitido, estamos a falar de situações que ocasionarão um erro de leitura e processamento em sua aplicação.

Eu sua IDE, ao tentar redigir alguma dessas variáveis de exemplo, serão acionados alguns alertas lhe notificando que esta sintaxe não é permitida. Ignorando estes alertas e tentando executar o código, será gerado um erro e a aplicação será encerrada.

No exemplo, logo de início temos uma variável que tem seu nome iniciado com número, seguido de uma segunda variável de nome inteiramente numérico. Também temos uma variável com nome iniciado com caractere especial. Por fim temos uma variável com seu nome encapsulado entre parênteses. Todas estas formas caem neste mesmo contexto de sintaxe não permitida.

```
and       del       from      not       while

as        elif      global    or        with

assert    else      if        pass      yield

break     except    import    print

class     exec      in        raise

continue  finally   is        return

def       for       lambda    try
```

Por fim, na imagem acima estão listadas as principais palavras reservadas ao sistema. Estas palavras são na verdade marcadores para que o interpretador identifique alguma estrutura ou funcionalidade padrão do núcleo da linguagem.

Por conta disto, usar de alguma destas palavras como nome para uma variável também irá gerar um erro na execução do programa.

Apenas contextualizando, a palavra reservada while é o gatilho para uma estrutura de código baseada em ciclos de repetição, executando certas ações, algo que veremos em seu devido momento neste livro.

```
1  nome1 = 'Fernando'
2
3  nome1 = 'Carine'
4
```

Outro ponto interessante a destacar é que, uma vez declarada uma variável, com um certo dado / valor atribuído, podemos utilizar livremente da mesma, inclusive atualizando / modificando seu conteúdo.

No exemplo acima, a variável chamada nome1 inicialmente recebe como atributo ‘Fernando’. Já na linha 3 deste código, uma nova instancia da variável nome1 é inserida, desta vez recebendo como atributo ‘Carine’. De acordo com a leitura léxica, haja visto que a leitura por parte do interpretador ocorre de

maneira sequencial, o valor lido para esta variável, último valor lido para nome1 será ‘Carine’.

Portanto devemos tomar cuidado no processo de declaração e manipulação de variáveis pois as mesmas podem assumir diferentes comportamentos ao longo da execução de um programa.

ENTRADA E SAÍDA DE DADOS

Outro ponto fundamental a ser entendido neste momento é o mecanismo do qual utilizaremos tanto para interagir com o usuário, permitindo que o mesmo dê entrada em dados na aplicação, quanto que após alguma ação realizada pela mesma, algum retorno seja exibido em tela ao usuário.

Cabe salientar que neste primeiro momento, estaremos interagindo com o usuário através de um terminal. Aplicações mais elaboradas requerem toda uma etapa de criação de interface gráfica com o usuário, aqui, estaremos nos atendo aos mecanismos internos que permitirão este tipo de interação.

Quando estamos trabalhando com a linguagem de programação Python, basicamente temos uma função de entrada chamada input( ) e uma outra função, esta de saída, chamada print( ).

Através de input( ) poderemos fazer com que o usuário digite algo diretamente via terminal, atribuindo este dado / valor a uma variável.

Dentro da mesma lógica, uma vez que temos algum dado / valor associado a uma variável, podemos exibir em tela para o usuário tal conteúdo através de uma função print( ).

```
1   nome = input('Digite o seu nome: ')
2
3   print('Bem vindo(a)', nome)
4
```

Partindo para a prática, na primeira linha de nosso código é declarada uma variável nomeada nome, que chama a função input( ), por sua vez parametrizada com uma mensagem de orientação ao usuário. *Posteriormente vamos entender a fundo o motivo em particular pelo qual tal mensagem deve ser escrita dessa forma, entre ‘ ‘ aspas. Por hora, apenas entenda que é possível “parametrizar” a função input( ) com uma mensagem que será exibida ao usuário quando nossa aplicação for executada.

Retomando nossa linha de raciocínio, a partir do momento que o usuário tenha digitado seu nome e pressionado ENTER, tal nome estará devidamente atribuído a variável nome.

Por fim, inserindo diretamente no corpo de nosso código uma função print( ), por sua vez parametrizada com uma mensagem e a instância da variável nome, ao executar este código será exibido em tela tal mensagem personalizada ao usuário.

Executando este código, o retorno será:

Digite o seu nome: *Fernando*

Bem vindo(a) Fernando

Em um primeiro momento, é exibido via terminal a mensagem “*Digite o seu nome:* “ ao usuário. Digitado o nome e pressionado ENTER, imediatamente após esta ação é exibida em tela ao mesmo a mensagem “*Bem vindo(a) Fernando*”.

Como veremos ao longo dos próximos capítulos, existem formas mais elaboradas de se trabalhar com tais funções. Por hora, apenas para que não tenhamos limitações para seguir com os próximos exemplos, isto é suficiente.

TIPOS DE DADOS

Avançando alguns passos em nossos estudos, vamos buscar entender os tipos de dados suportados pela linguagem Python e sua utilização.

De modo geral, a linguagem Python é uma linguagem “de tipagem fraca”, totalmente dinâmica, de modo que de acordo com a sintaxe aplicada sobre uma variável, definimos o tipo de dado da mesma.

Por exemplo, imagine que estamos a atribuir um dado 12345 a uma variável de nome numeros, nesta notação, numeros é uma variável de tipo int / inteiro, alterando 12345 para ‘12345’, apenas pelo fato da inserção de “aspas”, o interpretador passará a ler a variável numeros como de tipo string / texto. Isto implica no modo de iteração de outras estruturas de código com o conteúdo desta variável, pois, apenas a nível de exemplo, uma operação matemática não pode ser aplicada a um texto...

Existem alguns tipos de dados básicos em Python, suficientes para que possamos elaborar 95% de nossas estruturas de código. Para casos bastante específicos, existem bibliotecas que implementam novas funcionalidades ao núcleo da linguagem, incluindo outros tipos de dados com suas respectivas particularidades.

Basicamente, dos tipos de dados padrão em Python temos:

int – Números de tipo inteiro

float - Números com casas decimais

string – Texto composto por qualquer caractere alfanumérico

tupla – Conjunto de elementos de diferentes tipos, imutável

list – Listas de elementos indexados e dinâmicos

dict – Dicionários estruturados em chave e valor

bool – Booleano (verdadeiro ou falso)

Agora, vamos buscar entender as particularidades de cada um destes tipos de dados:

Tipo int

```
1   num1 = 2001
2
3   print(num1)
4
5   print(type(num1))
6
```

Inicialmente, é declarada uma variável de nome num1, que recebe como atributo o dado numérico 2001. Pela notação que pode ser observada no exemplo, basta digitar tal número diretamente que o mesmo será assim imediatamente reconhecido pelo interpretador como número de tipo inteiro.

A partir de uma função print( ) a qual nesse contexto é utilizada para exibir em tela, quando nossa aplicação for executada, o conteúdo da variável num1.

De modo parecido, repassando como parâmetro para a função print( ) uma outra função chamada type( ), por sua vez parametrizada com a variável num1, estaremos exibindo em tela o tipo de dado reconhecido para a variável num1.

Neste caso, ao executar este código, o retorno será:

2001

<class 'int'>

Na primeira linha desse retorno, nos é exibido em tela o conteúdo da variável num1, neste caso, 2001.

Já na segunda linha do retorno gerado temos <class ‘int’> que é o tipo de dado da variável num1, uma classe de tipo inteiro.

Tipo float

```
1   num1 = 19.90
2
3   print(num1)
4
5   print(type(num1))
6
```

De modo parecido como para o exemplo anterior, quando estamos a falar de números, existe o formato chamado float, onde representamos números compostos de casas decimais.

Note que o símbolo utilizado para separação da parte inteira da parte decimal do mesmo número é um ponto. Isto se dá por conta do símbolo de vírgula ter outros propósitos específicos.

Nos mesmos moldes como feito para o exemplo anterior, através das funções print( ) e print(type( )) exibimos em tela o conteúdo da variável num1 e seu tipo de dado, respectivamente.

Neste caso, não havendo nenhum erro de sintaxe, ao executar o código o retorno será:

19.90

<class 'float'>

Tipo string

Avançando mais alguns passos, se tratando do tipo de dado chamado string, estamos a trabalhar com toda e qualquer representação de texto atribuído a uma variável.

Por parte de sintaxe, tudo o que estiver envolvido entre ‘’ aspas simples ou “” aspas duplas será reconhecido pelo interpretador como um texto (inclusive números representados desta forma).

```
1   nome = 'Fernando'
2
3   print(nome)
4
5   print(type(nome))
6
```

A nível de exemplo, declarada uma variável nomeada apenas nome, a mesma recebe de acordo com a notação a string ‘Fernando’.

Exatamente como feito anteriormente para os demais tipos de dados, exibimos em tela o conteúdo e tipo de dado da variável nome.

Neste caso, o retorno será:

Fernando

<class 'str'>

```
1   nome = 'Fernando'
2   nome2 = "Maria"
3   palavra1 = "Marca d'água"
4
5   print(nome)
6   print(nome2)
7   print(palavra1)
8
```

Explorando outras possibilidades, no exemplo acima é reaproveitada a variável nome e logo em seguida declaramos outra variável chamada nome2. Repare que nome2 por sua vez recebe a string “Maria”, na notação de aspas duplas.

Já para situações onde temos de utilizar ambos tipos de aspas em um mesmo texto, podemos utilizar de aspas duplas para definir a string e de aspas simples no lugar da apóstrofe. Como exemplo, na linha 3 de nosso código temos uma variável chamada palavra1 que recebe como atributo a string “Marca d’água”.

Executando este código, o retorno será:

Fernando

Maria

Marca d'água

```
1   nome = 'Fernando'
2   nome2 = "Maria"
3   palavra1 = "Marca d'água"
4   frase1 = 'Python é uma linguagem de "alto nível"'
5
6   print(nome)
7   print(nome2)
8   print(palavra1)
9   print(frase1)
10
```

Nesta mesma lógica, a combinação de aspas simples e duplas pode ser utilizada quando queremos dar destaque a alguma palavra específica da string.

Na linha 4 de nosso código é declarada uma nova variável, dessa vez nomeada frase1, que recebe a string ‘Python é uma linguagem de “alto nível”’. Note que a sintaxe aplicada aqui dá ênfase as palavras alto nível.

O retorno será:

Fernando

Maria

Marca d'água

Python é uma linguagem de "alto nível"

```
1   codigo1 = '887237'
2   codigo2 = 887237
3
4   print(codigo1)
5   print(codigo2)
6
7   print(type(codigo1))
8   print(type(codigo2))
9
```

Apenas fazendo um pequeno adendo, cabe salientar mais uma vez que os dados atribuídos a uma variável serão identificados de acordo com a sintaxe aplicada.

Por exemplo, no código acima a variável codigo1 recebe um número em forma de string, enquanto a variável codigo2 recebe o mesmo valor em forma numérica. Na prática, os dados atribuídos à codigo1 serão lidos apenas como texto, enquanto os dados atribuídos a variável codigo2 serão identificados como números, permitindo inclusive operações matemáticas sobre tais valores.

Executando este código, como esperado, o retorno será:

887237

887237

<class 'str'>

<class 'int'>

Tipo lista

Até o momento, buscamos entender alguns dos tipos de dados básicos que compartilhavam de uma característica básica, ser “um” dado / valor atribuído a “uma” variável. Mas e quando temos de atribuir mais dados a uma variável? Bem, nestes casos, temos de gerar um “contêiner” de dados, que do mesmo modo para como os demais tipos de dados, de acordo com sua sintaxe, pode ser um diferente tipo de dado.

Em Python, basicamente temos alguns tipos de contêineres de dados, cada um com suas particularidades. O tipo mais comum é o que convencionalmente chamamos de listas, um objeto de sintaxe própria em notação de colchetes [ ] que guarda em seu interior como elementos próprios outros tipos de dados respeitando suas respectivas sintaxes, os separando com vírgula, de modo que tais dados possuem um valor de indexação que utilizaremos para iterar sobre os mesmos.

Muito além de simplesmente guardar múltiplos dados indexados associados a uma só variável, como veremos em seguida, a partir deste tipo de dado será possível aplicar uma série de funções sobre seus elementos.

```
1   lista1 = [2, 4, 6, 8, 10]
2
3   print(lista1)
4
5   print(type(lista1))
6
```

Partindo para a prática, na primeira linha de nosso código é declarada uma variável chamada lista1 que recebe como atributo uma lista composta por 5 elementos de tipo numérico inteiro.

Do mesmo modo como feito para os exemplos anteriores, repassando uma instância da variável lista1 para uma função print( ) exibimos em tela seu conteúdo, assim como via print(type( )) exibimos seu tipo de dado.

Neste caso, o retorno será:

[2, 4, 6, 8, 10]

<class 'list'>

Na primeira linha do retorno gerado, temos o conteúdo de lista1, que será representado inclusive com seus colchetes e vírgula como separador de seus elementos.

Na última linha do retorno temos o tipo de dado, neste caso, uma classe que representa o tipo de dado list.

```
1   lista1 = [2, 4, 6, 8, 10]
2
3   print(lista1)
4
5   print(lista1[3])
6
```

Como mencionado anteriormente, os elementos de uma lista são dados / valores indexados. Índices em Python no geral são representações que iniciam em 0, seguindo de modo crescente para seus elementos subsequentes.

```
              0, 1, 2, 3,  4
1   lista1 = [2, 4, 6, 8, 10]
2
3   print(lista1)
4
5   print(lista1[3])
6
```

Sendo assim, aplicando esta notação implícita sobre os elementos de lista1, sabemos que o elemento de índice 0 é 2, o elemento de índice 1 é 4, o elemento de índice 2 é 6, o elemento de índice 3 é 8, por fim, o elemento de índice 4 é 10.

A partir disto, utilizando da notação de “fatiamento”, representada pelo uso de [ ] colchetes sobre a instância de uma variável, podemos selecionar um elemento específico ou vários elementos dentro de um intervalo através de seu índice.

Como exemplo, na linha 5 de nosso código temos uma função print( ) por sua vez parametrizada com lista1[3]. Em outras palavras, aqui estamos selecionando apenas o elemento de índice 3 para ser exibido em tela via função print( ).

Executando este código, o retorno será:

[2, 4, 6, 8, 10]

8

Na primeira linha do retorno, todo o conteúdo de lista1.

Na segunda linha do retorno, apenas o elemento situado na posição de índice 3, neste caso, o valor inteiro 8.

```
1   lista1 = [2, 4, 6, 8, 10]
2   lista2 = ['Python', 2001, 19.90, [1, 2, 3]]
3
4   print(lista1)
5   print(lista2)
6
7   print(lista1[3])
8   print(lista2[0])
9   print(lista2[1:3])
10
```

No enunciado deste tópico, fora comentado que uma lista comporta em seu interior qualquer tipo de dado, desde que respeitado sua sintaxe.

A nível de exemplo, na segunda linha de nosso código declaramos uma nova variável chamada lista2 que recebe como atributo uma lista de variados tipos de elementos, sendo de acordo com sua sintaxe uma string na posição de índice 0, seguido de um valor de tipo int, um valor float, por fim uma lista de 3 elementos.

Posteriormente veremos que é possível inclusive iterar sobre os elementos de listas dentro de listas ou demais tipos de contêineres de dados.

Na linha 8 de nosso código, de acordo com a notação, estamos buscando exibir em tela apenas o elemento de índice 0 de lista2.

Já na linha 9 deste código temos outra situação específica, onde utilizando da seguinte notação [1:3], estaremos selecionando os elementos dentro de um intervalo de índice específico, nesse caso, a partir do elemento de índice 1 até o último elemento antes do índice 3. Em Python temos esta particularidade que inicialmente pode ser um tanto quanto confusa, mas é a forma como o interpretador encontra para saber qual será o marcador final do intervalo de seleção. Raciocine que neste caso [1:3] retornará apenas os elementos de índice 1 e 2, pois este valor 3 é o indicador para o interpretador do término da seleção e apenas isto, não retornando este elemento na própria seleção.

Neste caso em particular, executando este código o retorno será:

[2, 4, 6, 8, 10]

['Python', 2001, 19.9, [1, 2, 3]]

8

Python

[2001, 19.9]

Na primeira linha do retorno, todo o conteúdo de lista1.

Na segunda linha do retorno, todo o conteúdo de lista2.

Na terceira linha do retorno, apenas o elemento situado na posição de índice 3 de lista1.

Na quarta linha do retorno, apenas o elemento de posição de índice 0 de lista2.

Por fim, na quinta linha do retorno, os elementos situados dentro do intervalo 1-3 de lista2.

Tipo tupla

Uma tupla, contêiner de dados com este nome um tanto quanto diferente, é muito semelhante a uma lista, também responsável por armazenar elementos indexados, porém com uma particularidade. Enquanto uma lista é um tipo de dado dinâmico, onde (como veremos nos capítulos seguintes) podemos iterar sobre seus dados modificando os mesmos, uma tupla armazena seus dados de modo imutável, e isto é útil para cenários onde de fato dados / valores devem permanecer inalterados independentemente de sua utilização.

A nível de sintaxe, ao encapsular elementos entre ( ) parênteses, estamos fazendo com que este contêiner de dados seja imediatamente identificado pelo interpretador como de tipo tupla.

```
1   tupla1 = (2, 4, 6, 8, 10)
2   tupla2 = ('Python', 2001, 19.90, [1, 2, 3])
3
4   print(tupla1)
5   print(tupla2)
6
7   print(tupla1[3])
8   print(tupla2[0])
9   print(tupla2[1:3])
10
```

A nível de exemplo, replicamos o mesmo conteúdo das listas do exemplo anterior, apenas alterando os devidos nomes de suas respectivas variáveis e a notação de colchetes para parênteses.

Executando este código, o retorno será exatamente o mesmo obtido como para o exemplo anterior, haja visto que as mesmas operações de seleção via notação de fatiamento também se aplicam a este tipo de dado.

Tipo dicionário

Avançando alguns passos em nossos estudos, chegamos em um tipo de dado muito interessante chamado usualmente de dicionário. Este tipo de dado, além de uma sintaxe própria representada por { } chaves, possui uma estrutura interna composta por “chaves” e “valores”, sempre inseridos em conjunto e separados por “ : “, de modo que sempre quando iteramos sobre uma determinada chave, recebemos seu respectivo dado / valor.

```
1  clientes = {'Ana': 'anasilva01@gmail.com',
2              'Maria': 'mari1987@outlook.com.br',
3              'Pedro': 'p.santos.22@live.com'}
4
5  print(clientes)
6
7  print(clientes['Maria'])
8
```

Para ficar mais claro, nada melhor que partir para o exemplo. Em nosso código, é declarada uma variável de nome clientes, que recebe como atributo um dicionário composto de 3 elementos.

Observe bem a sintaxe, o primeiro elemento possui a chave ‘Ana’ e seu respectivo valor ‘anasilva01@gmail.com’. Caso fossem outros dados, os mesmos estariam em sua devida sintaxe, desde que sempre respeitando a forma de chave:valor exigida para um dicionário.

Parametrizando uma função print( ) com a variável clientes, exibimos em tela seu conteúdo.

Já repassando como parâmetro para print( ) a variável clientes, referenciando uma chave, neste caso ‘Maria’, receberemos como retorno seu respectivo valor.

Neste caso, o retorno será:

{'Ana': 'anasilva01@gmail.com', 'Maria': 'mari1987@outlook.com.br', 'Pedro': 'p.santos.22@live.com'}

mari1987@outlook.com.br

Na primeira linha do retorno, todo o conteúdo do dicionário associado a variável clientes.

Na segunda linha do retorno, apenas o valor referente a chave ‘Maria’ do dicionário atribuído a clientes.

```
clientes = {'Ana Silva': [22, 'Professora'],
            'Maria Santos': [56, 'Advogada'],
            'Pedro Pereira': [15, 'Estudante']}

print(clientes['Ana Silva'])

```

Apenas para fins de exemplo, tornando mais robusta a estrutura de dados associada à variável clientes, podemos perfeitamente utilizar de um dicionário onde os valores são listas, tuplas ou até mesmo dicionários. Neste caso, cada chave possui uma lista de elementos como valor, sempre respeitando sua sintaxe.

Executando este código, o retorno será:

[22, 'Professora']

```
clientes = {'Ana Silva': [22, 'Professora'],
            'Maria Santos': [56, 'Advogada'],
            'Pedro Pereira': [15, 'Estudante']}

print(clientes['Ana Silva'][1])

```

Para iterar sobre um elemento específico, da lista-valor associada a uma determinada chave, utilizamos também da notação de fatiamento.

Supondo que queremos obter como retorno apenas o elemento de índice 1 da lista associada a 'Ana Silva', aplicamos o fatiamento baseado em dois colchetes, sendo o primeiro instanciando a chave, neste caso ['Ana Silva'] seguido de um segundo colchete com a posição de índice do elemento a ser extraído informação.

Neste caso, executando o código, o retorno será:

Professora

Tipo booleano

Outro tipo de dado bastante particular é o chamado booleano, que basicamente é utilizado para abstrair e representar valores lógicos atribuídos a uma variável.

Em outras palavras, este tipo de dado será basicamente representado pela sintaxe True ou False, indicando assim um estado "verdadeiro" ou "falso" a partir de uma variável.

Posteriormente veremos em detalhes as chamadas estruturas condicionais, onde através de proposições lógicas determinaremos certos desfechos para certas funcionalidades de nosso código, utilizando como base este tipo de dado.

```
1   valor1 = True
2   valor2 = False
3
4   print(valor1 and valor2)
5
6   print(10 > 20)
7
```

Apenas contextualizando, pois de fato veremos este tipo de dado em ação apenas em capítulos futuros, observe o exemplo.

Para as variáveis valor1 e valor2 são atribuídos valores booleanos True e False, respectivamente. Inclusive note que estas são palavras imediatamente identificadas como reservadas ao sistema.

Exibindo em tela a relação entre valor1 e valor2, como em uma tabela verdade, para True / Verdadeiro e False / False é esperado False.

De modo parecido, ao se tentar executar alguma operação lógica ou matemática dentro de uma função print( ), o retorno será um valor booleano. Em nosso exemplo, validando se 10 é maior que 20, o retorno gerado será False.

OPERADORES

Os chamados operadores basicamente são caracteres especiais que atuam como mini funções específicas, normalmente categorizados entre operadores lógicos ou numéricos ou relacionais, desempenhando papel fundamental para a interação entre variáveis.

Existe uma vasta gama de operadores que podem ser empregados para de modo reduzido e simplificado realizar operações sobre os dados de variáveis.

Operadores matemáticos

Começando pelo princípio, os operadores matemáticos, como próprio nome sugere, são objetos básicos que permitem a realização de funções matemáticas sem a necessidade de as programar do absoluto zero.

Basicamente, através de marcadores / caracteres especiais reservados ao sistema, podemos iterar sobre os valores de variáveis realizando os cálculos necessários para um propósito específico.

```
1   valor1 = 43
2   valor2 = 58
3
4   print(valor1 + valor2) # Soma
5   print(valor1 - valor2) # Subtração
6   print(valor1 * valor2) # Multiplicação
7   print(valor1 / valor2) # Divisão
8
9   print(valor1 + valor2 * 3) # Mais de uma operação simples
10  print((valor1 + valor2) * 3) # Operação composta
11
12  print(valor1 ** valor2) # Potenciação
13  print(valor1 // valor2) # Divisão exata
14  print(valor1 % valor2) # Módulo / resto de uma divisão
15
```

Em nosso exemplo, logo nas primeiras linhas são declaradas duas variáveis de nomes valor1 e valor2 que recebem como atributo dois valores numéricos inteiros 43 e 58, respectivamente.

A partir disto, podemos realizar operações sobre estes valores, exibindo em tela o produto desta operação ou até mesmo atribuindo este resultado para uma nova variável.

Em nosso código, de maneira simplificada realizamos operações de soma, subtração, multiplicação e divisão através de seus respectivos operadores +, -, * e /.

Nas linhas 9 e 10 do código, um exemplo para demonstrar que, se tratando de operações matemáticas, as mesmas regras para todo e qualquer cálculo aqui

também se aplicam. Por exemplo, em operações com mais de dois operandos, as multiplicações e divisões são via regra executadas antes das somas e subtrações. Para contornar este quesito e "forçar" uma operação a ser executada antes da regra geral, se faz necessário encapsular os operandos envolvidos em parênteses.

Em outras palavras, na operação apresentada na linha 9, inicialmente será multiplicado o valor de valor2 por 3 e apenas após isto tal valor será somado ao valor de valor1. Já para a mesma operação situada na linha 10, de acordo com a notação aplicada, inicialmente serão somados os valores de valor1 e valor2, tendo este valor resultante multiplicado por 3.

Também é possível observar no exemplo que existem operadores até mesmo para situações bastante específicas, como por exemplo potenciação, divisão exata (onde não sobra resto) ou módulo / resto de uma divisão não inteira.

Rodando este código, não havendo nenhum erro de sintaxe, o retorno será:

101

-15

2.494

0,7413793103448276

217

303

73885357344138503765449

3

1

Operadores de atribuição

Ao longo de todos os exemplos anteriores, utilizamos do operador de atribuição " = " para associar dados / valores às suas respectivas variáveis.

```
valor1 = 43 # Atribuição simples

print(valor1)

valor1 += 5 # Atribuição aditiva
# Mesmo que: valor1 = valor1 + 5
# Mesmo que: valor1 = 43 + 5

valor1 -= 5 # Atribuição subtrativa
# Mesmo que: valor1 = valor1 - 5
# Mesmo que: valor1 = 43 - 5

valor1 *= 2 # Atribuição multiplicativa
# Mesmo que: valor1 = valor1 * 2
# Mesmo que: valor1 = 43 * 2

valor1 /= 2 # Atribuição divisiva
# Mesmo que: valor1 = valor1 / 2
# Mesmo que: valor1 = 43 / 2

```

Avançando alguns passos, é válido entender que existem operadores de atribuição que de forma reduzida permitem realizar operações matemáticas sobre os valores iniciais de uma variável. Na prática, pegamos o valor inicial de uma variável, realizamos algum cálculo sobre tal valor onde ao final do processo o valor resultante sobrescrevesse / atualizasse o valor inicial da própria variável.

Desta forma, a partir dos operadores +=, -=, *= e /= podemos atualizar o conteúdo de uma variável com operações matemáticas aplicadas a outro valor explícito ou oriundo de outra variável.

Neste caso, o retorno será:

43

48

38

86

21.5

Operadores lógicos

```
valor1 = 43
valor2 = 14

print(valor1 == valor2) # Operador de igualdade
print(valor1 == 43)

print(valor1 != valor2) # Operador de diferença

print(valor1 > valor2) # Maior que
print(valor1 >= valor2) # Igual ou maior que
print(valor1 < valor2) # Menor que
print(valor1 <= valor2) # Igual ou menor que

```

Os chamados operadores lógicos são os que de alguma forma nos permitem realizar comparações entre os dados / valores de duas ou mais variáveis. Nesse contexto, é possível verificar se o valor de uma variável é exatamente o mesmo valor de outra via operador ==, ou se é diferente via !=.

Para as comparações diretas, através dos operadores >, >=, < e <= validamos se os valores de duas variáveis são, em sua ordem de declaração na expressão lógica (valor1 em relação a valor2), maior que, igual ou maior que, menor, igual ou menor que.

False

True

True

True

True

False

False

Observe que como resposta para cada proposição lógica, nos é retornado um valor booleano. Pegando como exemplo o resultado da primeira proposição lógica, situada na linha 4 de nosso código, obtemos False pois o valor de valor1 não é igual ao valor de valor2. A mesma regra se aplica a todas as demais proposições, retornando verdadeiro ou falso conforme a comparação realizada.

Operadores relacionais

No tópico anterior, referente aos operadores lógicos simples, utilizamos de exemplo algumas proposições simples, baseadas em comparação direta dos valores de uma variável em relação a outra. Porém, é perfeitamente possível realizar validações sobre operações lógicas compostas, sempre seguindo como regra a tabela lógica simples AND.

```
1   valor1 = 43
2   valor2 = 14
3
4   print(valor1 == valor2)
5
6   print(valor1 > 40 and valor1 >= valor2) # Operador AND / E
7   # valor1 é maior que 40 e valor 1 é igual ou maior que valor2?
8
9   print(valor1 > 40 or valor1 < valor2) # Operador OR / OU
10  # valor1 é maior que 40 ou valor1 é menor que valor2?
11
```

Utilizando do operador and, apenas receberemos como retorno True caso as duas expressões lógicas em questão também retornarem True. No exemplo, o retorno para a primeira proposição (valor1 > 40) é True, assim como o valor retornado pela segunda proposição (valor1 >= valor2) também é True, logo, seguindo o princípio da tabela verdade AND, True e True resulta True.

Em contrapartida, através do operador or bastando uma das proposições serem validadas como verdadeira, já é retornado True para toda a expressão. No exemplo, o retorno da primeira proposição é True, enquanto o retorno da segunda proposição é False, uma vez que o valor de valor1 não é menor que o valor de valor2, ainda assim, para esta estrutura lógica será retornado True ao final do processo.

Para este exemplo, o retorno será:

False

True

True

```
valor1 = 43
valor2 = 14

print(valor1 == valor2)
print(valor1 is valor2) # Operador is
# O valor de valor1 é o mesmo valor de valor2?

lista1 = ['Ana', 48, 75.26, ['Professora', 'Diretora']]

print('Ana' in lista1) # Operador in
# 'Ana' consta dentro de lista1?

print('Ana' not in lista1) # Operador not in
# 'Ana' não consta dentro de lista1?

print('Ana' in lista1 or 'Maria' in lista1)
# 'Ana' consta em lista1 ou 'Maria' consta em lista1?

```

De modo parecido, utilizando dos operadores in ou not in podemos validar se um determinado dado / valor está presente em um contêiner de dados utilizado como referência.

A nível de exemplo, mais especificamente na linha 10 de nosso código, dada a proposição lógica ‘Ana’ in lista1 estamos a verificar se a string ‘Ana’ é um dos elementos da lista atribuída a variável lista1. Caso tal string seja um dos elementos, independentemente de sua posição de índice, será retornado True.

Já a utilização de not in aplica a chamada negação lógica, retornando True quando um determinado objeto inspecionado não fizer parte do contêiner de dados.

Por fim, sempre observando a correta sintaxe, realizando a combinação de todos estes operadores estudados até então acabamos pro ter um grande leque de possibilidades no que diz respeito a iteração sobre os dados de nossas variáveis.

Rodando este script, o retorno será:

False

False

True

False

True

ESTRUTURAS CONDICIONAIS

As chamadas estruturas condicionais, também chamadas de estruturas de controle de fluxo, basicamente são estruturas personalizadas onde impomos certas condições a serem validadas sobre os dados de uma ou mais variáveis, de modo que de acordo com a proposição certos desfechos podem ser tomados.

Estruturas condicionais em Python são criadas a partir dos marcadores / palavras reservadas ao sistema if, elif e else, que conforme veremos logo a seguir, possuem diferentes combinações dependendo do propósito.

Este é apenas o ponto de partida no que diz respeito ao uso de estruturas que conforme ação tomada pelo usuário desempenham uma determinada atividade, pois como veremos posteriormente, podemos combinar este tipo de estrutura de dados com outras estruturas (operadores lógicos / relacionais, por exemplo) para criar cadeias lógicas robustas.

```
1  nome = input('Digite o seu nome: ')
2
3  if nome == 'Fernando':
4      print('Bem vindo Fernando')
5  else:
6      print('Usuário desconhecido')
7
```

Indo direto ao ponto, como exemplo inicialmente declaramos uma variável chamada nome, que através da função de entrada input( ) pede que o usuário digite o seu nome conforme a instrução repassada ao mesmo.

Em seguida criamos uma estrutura condicional simples via if else, onde verificamos através da expressão “if nome == ‘Fernando’” (se o nome digitado pelo usuário, atribuído a variável nome é igual a ‘Fernando’ *incluindo a configuração de letras maiúsculas ou minúsculas), e se esta proposição for verdadeira, então é exibida em tela a mensagem ‘Bem vindo Fernando’. Caso contrário (else), é exibida a mensagem ‘Usuário desconhecido’.

Note que a nível de sintaxe, logo após if inserimos uma proposição lógica a ser validada, seguido de : dois pontos. Em uma nova linha de código, indentada a if,

programamos o desfecho a ser tomado, que pode ser literalmente qualquer ação. Já para else programamos o desfecho a ser tomado caso a proposição anterior não seja de nenhuma forma validada como verdadeira.

```
nome = input('Digite o seu nome: ')

if nome == 'Fernando':
    print('Bem vindo Fernando')
if nome == 'fernando':
    print('Bem vindo Fernando')
else:
    print('Usuário desconhecido')

```

Apenas como exemplo, mais de um if pode ser inserido em meio ao contexto de uma estrutura condicional. De acordo com a leitura léxica realizada por parte do interpretador, o mesmo irá testar sequencialmente cada um dos ifs, e a partir do momento em que encontrar uma proposição verdadeira, irá executar seu desfecho e encerrar a execução deste bloco de código.

Supondo que ao executar este código o usuário tenha digitado ‘Fernando’, o desfecho do primeiro if é executado e é encerrada a execução deste código.

Supondo que o usuário tenha digitado ‘fernando’, o primeiro if é testado e validado como False, pois ‘fernando’ é diferente de ‘Fernando’, já no segundo if a proposição é dada como verdadeira e seu desfecho é executado, finalizando a estrutura condicional neste ponto.

```python
nome = input('Digite o seu nome: ')

if nome == 'Fernando':
    print('Bem vindo Fernando')
elif nome == 'fernando':
    print('Bem vindo Fernando')
elif nome == 'Fernando Feltrin':
    print('Bem vindo Fernando')
elif nome == 'fernando feltrin':
    print('Bem vindo Fernando')
else:
    print('Usuário desconhecido')

```

Um passo além da simples combinação de if else, podemos utilizar de elif em situações onde mais de uma proposição pode ser verdadeira. Apenas como exemplo, no código acima realizamos de maneira bastante básica uma validação onde o nome digitado pelo usuário pode ser considerado verdadeiro dentro de quatro variações do mesmo, neste caso, validando a opção que se encaixa perfeitamente na notação utilizada pelo usuário em sua digitação.

```python
num = int(input('Digite um número entre 0 e 10: '))

if num == 0:
    print('Número Zero')
if num > 0 and num <= 5:
    print('Número entre 1 e 5')
if num == 5:
    print('Número Cinco')
if num > 5 and num <= 10:
    print('Número entre 6 e 10')

```

Utilizando de um exemplo numérico, vamos explorar outras possibilidades. No exemplo acima, note que há uma cadeia de validações realizadas via if e nenhum else, isto é perfeitamente possível quando não queremos um desfecho diferente dos programados.

O uso de múltiplos ifs acarreta em aceitar mais de uma proposição como verdadeira. Supondo que quando questionado o usuário tenha digitado 5, será

retornado o desfecho do segundo e do terceiro if, pois no segundo if é validado como verdadeiro que o valor 5 “é maior que zero e igual ou maior que 5”, enquanto no terceiro if o retorno verdadeiro se dá por conta de “o valor de num ser igual a 5”.

```
num = int(input('Digite um número entre 0 e 10: '))

if num == 0:
    print('Número Zero')
elif num > 0 and num <= 5:
    print('Número entre 1 e 5')
elif num == 5:
    print('Número Cinco')
elif num > 5 and num <= 10:
    print('Número entre 6 e 10')

```

Replicando o mesmo exemplo e alterando a cadeia lógica para uma combinação de if e elifs, assim que a primeira proposição desta cadeia for validada como verdadeira, o processamento deste bloco é encerrado e o interpretador parte para a leitura do bloco de código seguinte.

Neste caso, supondo que quando questionado o usuário tenha digitado o valor 5, apenas o desfecho do primeiro elif é inicializado, ignorando todos os demais mesmo que tenham mais proposições verdadeiras.

Das particularidades deste exemplo, cabe apenas uma ressalva sobre a primeira linha de código, onde aninhamos as funções int(input( )) para garantir que o número digitado pelo usuário seja devidamente reconhecido como de tipo numérico inteiro e não como um texto.

ESTRUTURAS DE REPETIÇÃO

Estruturas de repetição, como o próprio nome sugere, são estruturas de código responsáveis por manter em execução uma certa instrução por tempo indeterminado até que algum gatilho seja acionado para que tal processamento de dados seja devidamente encerrado.

Em Python existem duas formas básicas de criar estruturas de repetição, sendo uma através de um laço for, outra através de while. Basicamente, ambas atuam como uma estrutura condicional onde enquanto uma certa proposição for verdadeira, o ciclo de execução se repete.

```
1   nomes = ['Ana', 'Carlos', 'Mariana', 'Rafael']
2
3   for nome in nomes:
4       print(nome)
5
```

Como exemplo do laço de repetição for, inicialmente é declarada uma variável chamada nomes que recebe uma lista de 4 elementos nominais em forma de string.

Para criar um laço de repetição que irá iterar sobre cada um dos elementos de nomes, inserimos o marcador for, seguido de um nome para uma variável temporária (neste caso nome), do operador de membro in e da instância da variável a qual deverá iterar.

Indentado a este laço de repetição, apenas inserimos uma função print( ) que exibirá o último dado / valor atribuído para nome.

Vamos buscar entender a lógica aqui aplicada. Observe que, de acordo com a notação, um laço será aplicado sobre a variável nomes, a cada ciclo de processamento, o mesmo irá iterar sobre um dos elementos de nomes, atribuindo este dado / valor à variável nome, finalizando com a exibição em tela deste nome. Após esta ação ser finalizada, o laço itera novamente sobre nomes,

agora lendo o segundo elemento desta lista, atualizando a variável temporária nome com este dado / valor e o exibindo em tela. Este processo se repete até que todos os elementos de nomes tenham sido devidamente lidos e, apenas após isto, o ciclo de repetição é finalizado.

Executando este código, o retorno será:

Ana

Carlos

Mariana

Rafael

Na primeira linha de código temos o retorno da função print( ) parametrizada com o último dado / valor atribuído a variável temporária nome.

Na segunda linha temos como retorno o segundo elemento lido na lista nomes.

Na terceira linha o terceiro elemento e assim por diante, até que todos os nomes da lista tenham de fato sido exibidos em tela.

```
1   num = int(input('Digite um número: '))
2
3   while num <= 10:
4       print(num)
5       num += 1
6
```

Já a estrutura de repetição while costuma funcionar de um modo um pouco diferente. Aqui, programamos uma ação a ser executada até que um gatilho seja acionado encerrando o ciclo de repetição (gatilho necessário para que o bloco de código indentado a while não entre em um loop infinito de execução).

Como exemplo, inicialmente é declarada uma variável de nome num que recebe do usuário um número conforme a instrução.

Logo em seguida a estrutura while é inserida no corpo geral do código, validando uma proposição lógica onde “enquanto o valor de num for igual ou menos que 10”, então este número é exibido em tela via print( ) e como gatilho para o próximo ciclo, num tem seu valor atual atualizado, incrementado em uma unidade.

Supondo que quando questionado, o usuário tenha digitado 3, então o retorno será:

3

4

5

6

7

8

9

10

No retorno gerado, inicialmente é exibido em tela o próprio valor digitado pelo usuário, seguido do mesmo número atualizado / incrementado a cada ciclo de repetição até que seu valor é validado como False quando o mesmo for “maior que 10” de acordo com a expressão lógica.

Tenha em mente que nesta modalidade, tudo o que estiver programado como desfecho via while será executado até que o gatilho final seja acionado.

FUNÇÕES

Anteriormente vimos as funções de entrada e de saída input( ) e print( ), respectivamente, porém, é necessário entender que além de utilizar de funções pré-moldadas do núcleo da linguagem Python, podemos criar nossas próprias funções personalizadas.

Assim como em todas as demais estruturas de código vistas até então, o processo de se criar uma função personalizada possui uma sintaxe própria e uma lógica a ser seguida. Raciocine que uma função é criada justamente para realizar uma ou mais ações sobre os dados processados na aplicação, e isto pode ocorrer tanto a nível interno apenas manipulando dados quanto a nível externo, interagindo com o usuário de alguma forma.

Se tratando da sintaxe para esta estrutura de código, toda e qualquer função em Python é criada a partir do marcador def seguido de um nome personalizado e de um campo para inserção de parâmetros (caso houverem). Por fim, indentado ao corpo dessa função são inseridas as estruturas de código necessárias e programado um retorno a ser gerado quando a função encerrar seu processamento.

```
1  def exibe_mensagem():
2      print('Olá Mundo!!!')
3
4
5  exibe_mensagem()
6
```

Como exemplo básico, apenas para que possamos entender os conceitos apresentados anteriormente, logo na primeira linha de nosso código de acordo com a sintaxe aplicada criamos uma função chamada exibe_mensagem( ).

Note que a mesma possui um nome nos mesmos moldes como nomeamos uma variável, seguido de um campo de ( ) parênteses, onde podem ser inseridos parâmetros / argumentos / dados a serem processados por tal função.

Indentado ao corpo desta função, apenas inserimos uma função print( ) por sua vez parametrizada com uma mensagem em forma de string. Desta forma, sempre que “chamada” esta função, tal mensagem será exibida em tela, haja visto que esta é a única ação configurada para ser realizada por nossa função exibe_mensagem( ).

De volta ao corpo / escopo geral de nosso código, ao chamar a mesma, seu devido desfecho deverá ser inicializado.

Neste caso, executado o código, o retorno será:

Olá Mundo!!!

```
num1 = int(input('Digite o primeiro número: '))
num2 = int(input('Digite o segundo número: '))

resultado_soma = 0

def soma_dois_numeros(n1, n2):
    resultado_soma = n1 + n2
    print('O resultado é: ', resultado_soma)
    return resultado_soma

soma_dois_numeros(num1, num2)

```

A nível de exemplo um pouco mais elaborado, vamos criar uma simples função personalizada que recebe dois valores e retorna o produto de sua soma, buscando entender cada particularidade envolvida neste processo.

Para isso, inicialmente são declaradas duas variáveis nomeadas num1 e num2, que recebem do usuário valores conforme a mensagem de instrução.

Em seguida, apenas por convenção declaramos uma nova variável, dessa vez chamada resultado_soma, de valor inicial 0, valor este a ser atualizado posteriormente.

Eis que chegamos no bloco de código dedicado à nossa função personalizada. Aqui, definimos uma função chamada soma_dois_numeros( ) que de acordo

com a notação observada em seu campo de parâmetros / argumentos, quando inicializada deverá receber dados / valores para suas variáveis n1 e n2.

Dentro do corpo desta função, a variável resultado_soma é instanciada, recebendo como atributo o resultado da soma dos valores anteriormente repassados para n1 e n2.

Ainda nesse contexto, exibimos em tela o resultado desta soma através de uma função print( ).

Por fim, “retornamos” a instância de resultado_soma. Isto fará com que a variável resultado soma situada no corpo geral de nosso código, de valor 0, seja devidamente atualizada com o último valor obtido via função soma_dois_numeros( ).

Neste ponto, temos toda a estrutura necessária para que nossa função personalizada funciona corretamente. Próximo passo, “chamar” esta função para fazer uso da mesma.

Para isso, de volta ao corpo geral de nosso código inserimos uma instância da função soma_dois_numeros( ), repassando como parâmetros para a mesma os últimos valores atribuídos às variáveis num1 e num2.

Executando este código, supondo que quando questionado o usuário tenha digitado os valores 78 e 55, o retorno será:

Digite o primeiro número: *78*

Digite o segundo número: *55*

O resultado é: 133

CONSIDERAÇÕES FINAIS

Esta é uma obra para iniciantes em programação, porém, acredito que se você de fato pôs em prática os exemplos aqui referenciados, já deve ter entendido a lógica básica por trás de cada uma destas estruturas de dados.

Existe literalmente um universo de informações a serem buscadas para se aprofundar em cada um dos tópicos aqui abordados e explorar tudo o que esta magnífica linguagem de programação tem a oferecer, porém, o primeiro passo para a construção dessa bagagem de conhecimento se dá a partir deste tipo de material e destas estruturas de código por mais básicas que possam parecer.

A partir deste ponto, certamente você tem bases bastante sólidas para trilhar seus caminhos como desenvolvedor, uma jornada de constante prática e pesquisa em busca de aperfeiçoamento.

Quais são os próximos passos? Recomendo fortemente que você busque aprender sobre as particularidades e possibilidades oferecidas pelas ferramentas que Python nos proporciona, de modo a traçar um norte em direção a especialização para alguma área específica de desenvolvimento.

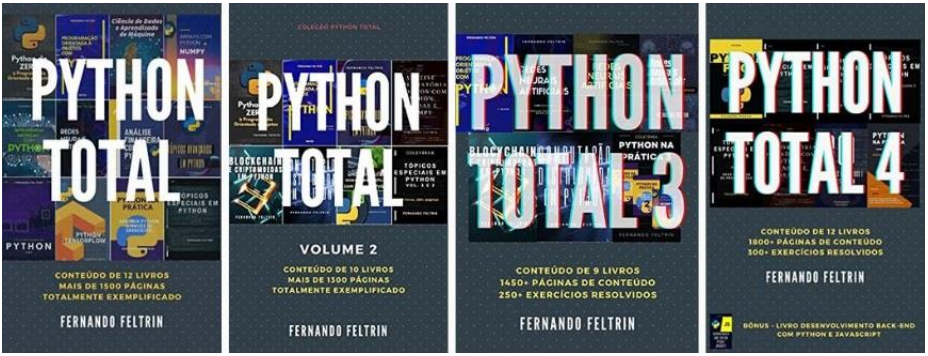
Por fim, resta meu sincero agradecimento por ter adquirido esta humilde obra e que a leitura da mesma tenha sido tão prazerosa a você quanto foi a mim por escrevê-la.

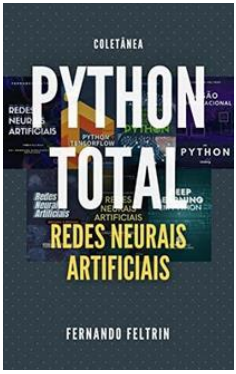
Um forte abraço.

Fernando Feltrin

PYTHON NEXUS

Coletâneas PYTHON TOTAL





Disponível em: [Amazon.com.br]

PYTHON NEXUS